Server Design

Srinidhi Varadarajan
Topics

- Types of servers
- Server algorithms
  - Iterative, connection-oriented servers
  - Iterative, connectionless servers
  - Iterative, connectionless servers
  - Concurrent, connection-oriented servers
- Server design issues
Need for Concurrency in Servers

- A simple server
  - Server creates a socket, binds address, and makes it passive
  - Server accepts a connection, services the request, the connection is closed, and this is repeated indefinitely

- Simple server is inadequate for most applications since the request may take arbitrarily long to service
  - Other clients are blocked from service
Concurrent versus Iterative Servers

- **An iterative server** services one request at a time
- **A concurrent server** services multiple requests at the same time
  - The actual implementation may or may not be concurrent
  - More complex than iterative servers
Three Dimensions of Server Design

- Iterative versus concurrent
  - Truly a server design issue as it is independent of the application protocol

- Connection-oriented versus connectionless
  - Usually constrained by the application protocol

- Stateless versus stateful
  - Usually constrained by the application protocol
<table>
<thead>
<tr>
<th></th>
<th>Connectionless</th>
<th>Connection-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>Iterative</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

- Concurrent, connection-oriented is the most common server design
1) Create a socket
   - sock = socket( PF_INET, SOCK_STREAM, 0 )

2) Bind to well-known address
   - bind( sock, localaddr, addrlen )
   - For port number, server can use
     getservbyname( name, protocol )
   - For host IP address, “wild card” address is usually used: INADDR_ANY

3) Place socket in passive mode
   - listen( sock, queuelen )
   - Need to establish queue length (maximum is implementation dependent)
Iterative, Connection-Oriented (2)

4) Accept a connection from a client
   - `new_socket = accept(sock, addr, addrlen)`
   - `accept()` blocks until there is at least one connection request
   - Based on the queue length value in `listen()`, connection requests may be “accepted” by the operating system and queued to be accepted later by the server with the `accept()` call

5) Interact with client
   - `recv(new_socket, ...)`
   - `send(new_socket, ...)`
Iterative, Connection-Oriented (3)

6) Close connection and return to accept() call (step 4)
   - close( new_socket )
     new_sock = accept(...)
     recv(new_sock,...)
     send(new_sock,...)
     close(new_sock)

other clients wait
Iterative, Connection-Oriented (4)

- Only one connection at a time is serviced by an iterative, connection-oriented server
  - Others wait in queue to be accepted
  - Or, their connection is refused
- TCP provides reliable transport, but there is overhead in making and breaking the connection
  - Simplifies application design
  - At the expense of a performance penalty
Iterative, Connectionless Server (1)

1) Create socket
   - `sock = socket( PF_INET, SOCK_DGRAM )`

2) Interact with one or more clients
   - `recvfrom(sock, buf, buflen, flags, from_addr, from_addrlen)`
     • Each subsequent `recvfrom()` can receive from a different client
     • `fromaddr` parameter lets server identify the client
   - `sendto(sock, buf, buflen, flags, to_addr, to_addrlen)`
     • `to_addr` is usually `from_addr` of preceding `recvfrom()`
Iterative, Connectionless Server (2)

sock=socket(…)

recvfrom(sock, …)

sendto(sock, …)

- Other clients block while one *request* is processed, not for a full connection time
- UDP is not reliable, but there is no connection overhead

response delay: other clients wait
Concurrent, Connectionless (1)

- Concurrency is on a *per request* basis for a connectionless server.
- There are two ways to achieve concurrency:
  - Create a new process, e.g. using `fork()` or `exec()`.
  - Create a new thread, using `pthread_create()`.
- "Master" thread uses `pthread_create()` to create a "slave" thread for each request.
Concurrent, Connectionless (2)

Master

M1) Create socket
   - sock = socket( PF_INET, SOCK_DGRAM )

M2) Read request
   - recvfrom(sock, ...)

M3) Create thread
   - pthread_create()
   - Thread knows:
     - IP address and port of client
     - Request information
     - Global data and socket

Return to M2
Concurrent, Connectionless (3)

Slave

S1) Respond to request
   – sendto(sock, …)

S2) Terminate
   – pthread_exit()
Concurrent, Connectionless (4)

MASTER

sock = socket(…)

recvfrom(sock, …)

thread_create()

SLAVE

sendto(sock, …)

pthread_exit()

SLAVE 2
**Concurrent, Connectionless (5)**

- Requests from multiple clients (or multiple requests from a single client) can be serviced concurrently
  - No long blocking periods
- `pthread_create()` does have overhead
  - Thread overhead can dominate if time to respond to request is small
  - Concurrent, connectionless server is a good design choice only if average processing time is long relative to thread overhead
- UDP offers no reliability, has no connection overhead
Concurrent, Connection-Oriented (1)

- Concurrency is on a *per connection* basis for a connection-oriented server
  - Depending on application, additional concurrency may also be possible
- There are three ways to achieve concurrency
  - Create a new process -- high overhead
  - Create a new thread -- lower overhead
  - Use *apparent concurrency* within a single thread
    - Lowest overhead
    - Based on select() call for *asynchronous* operation
Master, using thread

M1) Create socket
   - sock = socket( PF_INET, SOCK_STREAM )

M2) Bind address
   - bind(sock, ... )

M3) Put socket in passive mode
   - listen(sock, ... )
Master, using threads (continued)

M4) Accept a new connection
   - new_sock = accept(sock,…)

M5) Create thread
   - pthread_create()
   - Thread knows:
     • New socket -- new_sock
     • Global data

Return to M4
Concurrent, Connection-Oriented (4)

Slave, using threads

S1) Interact with client
   – recv(new_sock,...)
   – send(new_sock,...)

S3) Close socket
   – close(new_sock,...)

S2) Terminate
   – pthread_exit()
Concurrent, Connection-Oriented (5)

MASTER

new_sock=accept(…)

pthread_create()

SLAVE

recv(new_sock,…)

send(new_sock,…)
Concurrent, Connection-Oriented (6)

- Clients do not block while other clients are connected
  - One thread per client
  - Could have additional threads per client, but based on particular features of the application

- `pthread_create()` has overheads
  - Thread overhead can dominate if connection time is small
  - Concurrent, connection-oriented server is a good design choice only if average client connection time is long relative to thread overhead
Except on a true multiprocessor, “concurrency” from threads does not generally increase throughput!

- Transactions per second do not increase
- Delay for first service and variance for service time do decrease

Iterative:  

Concurrent:
May be able to increase throughput for some applications, e.g. by overlapping disk I/O with processing in the CPU

TCP provides reliability at the expense of connect/disconnect overhead
Apparent Concurrency (1)

0) Maintain a set of socket descriptors (SOCKETS) using the fd_set structure
   – Initialize SOCKETS = { } (empty)

1) Create socket
   – sock = socket( PF_INET, SOCK_STREAM )
   – SOCKETS = { sock }

2) Bind address
   – bind(sock, ... )

3) Put socket in passive mode
   – listen(sock, ... )
Apparent Concurrency (2)

4) Use select() to determine sockets that have activity (are ready for “service”)
   – ret = select(maxfd, rdfds, wrfds, exfds, time)

5a) If select() indicates main socket (sock) is ready, accept a new connection
   – new_sock = accept(sock,...)
   – SOCKETS = SOCKETS ∪ { new_sock }

5b) If select() indicates another socket (ready) is ready
   – recv(ready,...) to read request, and then
   – send(read,...) to send response

Return to step 4
Apparent Concurrency (3)

- While another connection is accepted or while one request from another client is serviced
- Clients do not wait full connection time
Apparent Concurrency (4)

- Data can be conveniently (or dangerously) shared between different clients
  - Not easy with multiple threads
Server Design Factors (1)

- **Time per request**
  - If high, a multithreaded design is best
  - If low, thread overhead may dominate performance and an iterative server or a server using apparent concurrency is best

- **Time per connection (connection-oriented)**
  - If high, a concurrent (threaded or apparent) server is best
  - If low, an iterative server is best

- **Number of active clients**
  - If high, concurrent server is best
  - If low, iterative server is best
Server Design Factors (2)

- **Overhead for thread creation**
  - Trade-offs for connection time and request response time are relative to thread creation time
  - Operating systems with low overhead thread creation increase opportunities to use multithreaded design

- **Need to share information between clients**
  - Easier in an iterative server or a server with apparent concurrency
  - More complex in a multithreaded server
Server Design Factors (3)

- LAN- versus WAN-based application
  - TCP’s reliability is more important in a WAN where packet loss and out-of-order delivery is more likely
  - LAN-based applications may be able to provide reliability with less “expense” using UDP than TCP

- Inherent reliability in the application
  - May eliminate the need to use TCP
Simple Deadlock

- Deadlock occurs when
  - Client is blocked waiting on server
  - Server is blocked waiting on client
- Simple example of server deadlock

```
SERVER                              CLIENT
accept() ←———► connect()
      ↓                  ↓
recv()                never_never_land()
```

Server is blocked waiting for data from the client 😞
More Subtle Deadlock (1)

- Deadlock may be much more subtle

SERVER

accept() → recv() → send() 

CLIENT

connect() → send(BIG_BUFFER) → Client blocks at send() since server is not receiving 😞

Server eventually blocks at send() since client never receives 😞
More Subtle Deadlock (2)

SERVER
recv()
blocked
send()
receive buffer

CLIENT
send()
client deadlock
recv()
receive buffer

server deadlock
Terminating a Connection (1)

- The application protocol determines when a connection should be closed
- Client may know when transaction is done
  - Examples:
    - FTP
    - HTTP 1.1 (persistent connections)
  - A “misbehaving” client can keep connections open, consuming server resources
  - Solutions
    - Time-out for the session (connect, idle, etc.)
    - Trusted clients
Terminating a Connection (2)

- Even if the server controls connection termination, there may still be problems
  - Operating system maintains connection information for $2 \times \text{MSL}$ (maximum segment life)
    - Allows OS to reject delayed, duplicate packets
    - Uses OS resources
  - Client can make lots of requests and consume resources faster than the server can free them

- Vulnerability to \textit{denial of service attacks}
Example: Threaded ECHO Server (1)

- Multiple-threaded concurrent, connection-oriented design

![Diagram](image_url)

- Master
- Slave

Socket for connect

Sockets for individual connections

SERVER
Example: Concurrent ECHO Server (2)

- Operation of concurrent ECHO server
  - `pthread_create()` called for each new connection
  - `TCPechod()` invoked for each thread
    - `recv()` and `send()` repeated until client closes the connection
    - Note that `TCPechod()` does *not* call `exit()` to exit the process if there’s an error -- just the thread terminates i.e. the thread calls `pthread_exit`.
    - Calling `exit` will terminate all threads and the process, a bad idea in this case
Example: Asynch ECHO Server (1)

- Single-thread concurrent, connection-oriented

![Diagram showing server and connections](image-url)
Example: Asynch ECHO Server (2)

- Uses select() call
  - select() indicates which sockets are ready for service
    - Input or connection for ECHO server
  - fd_set structures record the sets of sockets

```c
typedef struct fd_set {
    u_int fd_count;
    SOCKET fd_array[FD_SETSIZE];
} fd_set;
```
Example: Asynch ECHO Server (3)

- fd_set structures manipulated with macros
  - FD_CLR( fd, set ): remove fd from set
  - FD_SET( fd, set ): add fd to set
  - FD_ZERO( set ): empty set
  - FD_ISSET( fd, set ): test if fd is in set

FD_ZERO(&afds); // empty afds
FD_SET(msock, &afds); // add msock
Example: Asynch ECHO Server (4)

- **select()**
  - Checks all sockets in sets
    - set for input and connection request
    - set for output
    - set for exceptions
  - Blocks until at least one of the sockets is ready or time-out
  - Returns with the set changed to contain just the sockets ready for service

```c
select(FD_SETSIZE, &rfds,
      (fd_set *)0, (fd_set *)0,
      (struct timeval *)0)
```
Example: Asynch ECHO Server (5)

**Operation**
- Steps through all active sockets and checks to see if socket is ready
- Accepts a new connection and adds to set if master server socket (msock) is ready
- Calls echo() to echo new data if client connection socket is ready

**There may be several sockets ready for service**
You should now be able to …

- Identify the three dimensions of server design
- Identify factors and application requirements that affect design choice
- Select server design based on factors application requirements
- Design, implement, and test servers based on the four classes
- Recognize causes of deadlock